

# Operational semantics for signal handling

Maxim Strygin

School of Computer Science  
University of Birmingham

M.Strygin@cs.bham.ac.uk

Hayo Thielecke

School of Computer Science  
University of Birmingham

H.Thielecke@cs.bham.ac.uk

Signals are a lightweight form of interprocess communication in Unix. When a process receives a signal, the control flow is interrupted and a previously installed signal handler is run. Signal handling is reminiscent both of exception handling and concurrent interleaving of processes. In this paper, we investigate different approaches to formalizing signal handling in operational semantics, and compare them in a series of examples. We find the big-step style of operational semantics to be well suited to modelling signal handling. We integrate exception handling with our big-step semantics of signal handling, by adopting the exception convention as defined in the Definition of Standard ML. The semantics needs to capture the complex interactions between signal handling and exception handling.

## 1 Introduction

In operating systems, and specifically Unix and its descendants, signals provide a simple and efficient, if rather low-level, means of interprocess communication [10, 14, 16, 15, 3]. Put simply, a process can cause a branch of control in another process, causing it to run a signal handler in response to external events. A well known example is the `kill` signal telling a process to shut down (perhaps after first deallocating system resources, such as releasing memory).

Signals resemble exceptions in that control jumps to a handler that can be installed by the program. Nonetheless, there are some significant differences. Whereas exceptions typically abort from the context, in which they were thrown rather than returning to it, signal handlers resume control after they have run. Whereas exceptions are triggered at specific points by the code itself, signals arrive nondeterministically. In the literature on control constructs and their semantics, signals have received less attention than exceptions, and far less than first-class continuations.

Exceptions have become amenable to semantic analysis by a focus on their key control features, while abstracting away from implementation details and restrictions (such as the entanglement of exceptions in C++ with the class hierarchy and memory management by destructors). For instance, the exceptions monad [12] gives a highly idealized account of exceptions as functions  $A \rightarrow (B + E)$  that may either return normally with a  $B$  or raise an exception of type  $E$ .

The aim of the present paper is to address signal handling at a level of generality and abstraction comparable to that of other control constructs in the literature, idealizing where necessary and focusing on some key semantic features. Our motivation for defining such a semantics, and exploring different styles of definition, is to develop of a Hoare logic for signals. While program logic is beyond the scope of the paper, it is a reason for our investigating the big-step style of operational semantics. In big-step, a command  $c$  takes a pre-state  $s_1$  to a post-state  $s_2$  in a judgment of the form  $s_1, c \Downarrow s_2$ . This form of judgment is particularly convenient for proving the soundness of Hoare triples  $\{P\}c\{Q\}$ , since the precondition  $P$  refers to the pre-state  $s_1$  and the postcondition  $Q$  to the post-state  $s_2$  in a big-step judgement.

## Outline of the paper

We begin by reviewing the constructs that we will need, and how to define operational semantics for them, in Section 2. We then combine these constructs and define the semantics for the whole language in Section 3. To validate our definition, we examine how signal and exception handling interact in a series of examples in Section 4. As an alternative to big-step semantics, we define a small-step semantics as a stack machine in Section 5, and relate it to implementations. We compare the stack machine to the big-step semantics in Section 6. Section 7 concludes.

## 2 Language constructs

Before giving the formal definition of our operational semantics, we introduce the language constructs with their intended meaning, as well as design choices and simplifying assumptions. We start from a small imperative base language. This language has a standard semantics in terms of how a command  $c$  changes the state  $s_1$  into a new state  $s_2$ . In a big-step operational semantics, the form of such judgements is

$$s_1, c \Downarrow s_2$$

When the command  $c$  raises an exception  $e$  after producing the new state  $s_2$ , we write

$$s_1, c \Uparrow e, s_2$$

### Exceptions

The semantics of exceptions is fairly well understood, and it is greatly simplified by the fact that exceptions are block structured. The more primitive non-local jumps in C (given via the library functions `setjmp()` and `longjmp()`) would be much harder to formalize. Exception throwing and handling is easy to add to a big-step operational semantics. A classic example of such a semantics is the Definition of Standard ML [11], whose style we will follow.

In addition to the rules for the operations themselves, we also need to specify how the propagation of exceptions interacts with the other constructs of the language: this propagation will be done with the *exception convention* from the Definition of Standard ML. If the  $j$ -th premise of a big-step rule raises an exception, and the premises to its left do not, then the conclusion of the rule raises the same exception, and with the same state.

More precisely, suppose there is a big-step rule of the form

$$\frac{\dots c_1 \Downarrow s_1 \dots c_j \Downarrow s_j \dots c_n \Downarrow s_n}{\dots c \Downarrow s}$$

Then we implicitly extend this case to propagating exception by adding a rule

$$\frac{\dots c_1 \Downarrow s_1 \dots c_j \Uparrow e, s_j}{\dots c \Uparrow e, s_j}$$

To illustrate the exception convention, we consider how exceptions are propagated in a sequential composition  $c_1; c_2$ .

$$\frac{s_1, c_1 \Uparrow e, s_2}{s_1, (c_1; c_2) \Uparrow e, s_2} \quad \frac{s_1, c_1 \Downarrow s_2 \quad s_2, c_2 \Uparrow e, s_3}{s_1, (c_1; c_2) \Uparrow e, s_3}$$

Intuitively, the first command  $c_1$  may raise an exception, in which case the second command  $c_2$  has not run at all. Alternatively,  $c_1$  may terminate normally, and  $c_2$  may raise an exception. In either case, the combined command raises the same exception.

### Signals

The main construct we aim to address is signal handling. Signal handling is a form of interprocess communication, so that for full generality we would have to address the concurrent interaction between a signal sending and a signal handling process. To keep the semantics as simple as possible, we address only the *handling* part of the signal mechanism, while the truly concurrent interaction between sender and receiver is left for future work. Rather than modelling the signal sender explicitly, only the point of view of the process receiving the signals will be assumed, so that signals arrive nondeterministically, causing handlers to run unpredictably. In the authors' view, this focus on signal *handling* still presents sufficient programming and semantics challenges. First, the nondeterministic interference by signal handlers leads to the need to preserve resource invariants, much as interference between concurrent processes. Moreover, the assumptions a programmer can make about the delivery of signals are very weak, even if there is a specification of the sender's behaviour (which there usually is not). In the worst case, the signal sender may even be malicious, sending signals with the sole intent of causing damage via the actions of the signal handlers. In that sense, a nondeterministic sender is a worst-case but realistic assumption that the signal receiver has to be able to cope with.

As a language construct, signal handlers resemble both concurrency and exception handling. Our most significant idealization of signal handlers is directly inspired by exceptions in contrast to the unstructured `longjmp` that exceptions were designed to replace. We define an idealized block-structured form of signal handling in which a signal handler is installed at the beginning of the block and uninstalled at the end. It relates to `sigaction` the way exceptions related to `setjmp` and atomic synchronized blocks related to locking and unlocking.

For the operational semantics, we define a big-step semantics. This style of semantics appears particularly apt for the signals and exceptions kind of constructs. Essentially, the meaning of a block becomes a subtree of a larger derivation tree, which is convenient for keeping track of pre- and post-states. In the same way, the derivation tree of one-sided signal handler could be easily injected into a larger tree.

One may think of addressing one-sided interleaving with the same approach as complete interleaving. This is true to some extent, but there are important differences between them. The interaction between fully concurrent processes is symmetric, but there is no such symmetry between the signal body and the handler. Only the signal handler may interrupt the body, but not vice versa. This allows using a simpler approach for addressing signal handling. On the other hand, the general approach used for the fully concurrent interleaving might not be suitable, as the interaction is non-symmetric.

Figure 1 depicts the symmetric interleaving of concurrent processes compared to the one-sided interleaving of a process by its signal handlers. Dashed horizontal lines represent control flow; dotted vertical lines represent switches in the control flow due to interleaving. In both cases, the state  $\sigma_i$  that a process sees at some point could have been changed to some state  $\sigma_{i+1}$  by interleaved actions. These state changes need to be limited in some way, as otherwise no assumptions could be made by the process about the state, including resource invariants.

### One-shot and persistent signals

Signal handlers can have two different control flow semantics, which we call *persistent* and *one-shot*. A persistent signal handler can be run any number of times as long as it is installed. By contrast, a one-shot signal handler can be run at most once, as it becomes automatically uninstalled after being run the first time. In Unix, the system call for installing handlers takes a parameter that determines which of these behaviours is chosen.

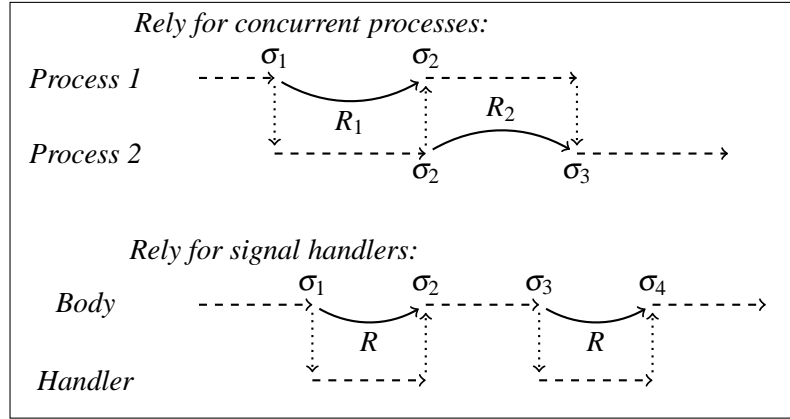


Figure 1: Processes vs signal handlers

### Operational semantics

In the operational semantics, the evaluation of a command  $c$  starting from a state  $s_1$  will now take place relative to a signal binding. Moreover, the signal binding is subdivided into two parts: persistent signals  $S$ , and one-shot signals  $O$ . Persistent handlers may run any number of times during the evaluation of the command  $c$ , whereas one-shot handlers may run at most once. The form of a big-step judgement with signal bindings is:

$$S; O \vdash s_1, c \Downarrow s_2$$

Note that the signal binding behaves like an environment (for variables bound via `let`) rather than a mutable state (for variables updated via `:=`). The judgement produces an updated state  $s_2$ , but it does not update  $S$  or  $O$ .

Analogous to binding an exception handler, we have two binding constructs for signals: one for persistent and one for one-shot handlers, where  $z$  is a signal name,  $c_b$  is a command, and  $c_h$  is a handler command.

$$\text{bind } z \text{ to } c_h \text{ in } c_b \quad \text{and} \quad \text{bind } /1z \text{ to } c_h \text{ in } c_b$$

To support signal disabling in a scope, we introduce two blocking constructs for signals:

$$\text{block } z \text{ in } c_b \quad \text{and} \quad \text{block } /1z \text{ in } c_b$$

Note that there is no need for an analogue of `throw  $e$`  (a command that throws an exception  $e$ ), as we assume that signals arrive nondeterministically from other, unspecified processes. The idea of using two contexts with a binder for each is loosely inspired by Barber and Plotkin's Dual Intuitionistic Linear Logic (DILL) [1].

## 3 Operational semantics for block-structured signals and exceptions

**Definition 3.1** The syntax of the language with signal and exception handling is given in Figure 2.

We let  $s$  range over states,  $c$  over commands,  $x$  over variables,  $v$  over values,  $e$  over exception names,  $z$  over signal names, and  $E$  over expressions, using subscripts where needed, e.g.,  $s_1$ ,  $e_2$ ,  $E_3$ ,  $c_4$  or  $c_h$ .  $s$  is a function from variables to values, such that  $s(x)$  returns a value  $v$ .

$c ::= \text{while}(E) \text{do } c$	(while construct)
$x := E$	(Assignment)
$c_1; c_2$	(Sequential composition)
$\text{throw } e$	(Exception throwing)
$\text{try } c_1 \text{ handle } e \text{ by } c_2$	(Exception handling)
$\text{bind } z \text{ to } c_1 \text{ in } c_2$	(Binding persistent signal handler)
$\text{bind}/1 z \text{ to } c_1 \text{ in } c_2$	(Binding one-shot signal handler)
$\text{block } z \text{ in } c$	(Blocking persistent signal)
$\text{block}/1 z \text{ in } c$	(Blocking one-shot signal)
$E ::= x \mid E + E \mid \dots$	(Expressions)

Figure 2: The syntax of the language

$\frac{S[z \mapsto c_1]; O \vdash s_1, c_2 \Downarrow s_2}{S; O \vdash s_1, \text{bind } z \text{ to } c_1 \text{ in } c_2 \Downarrow s_2}$	$\frac{S; O[z \mapsto c_1] \vdash s_1, c_2 \Downarrow s_2}{S; O \vdash s_1, \text{bind}/1 z \text{ to } c_1 \text{ in } c_2 \Downarrow s_2}$
$\frac{S - z; O \vdash s_1, c \Downarrow s_2}{S; O \vdash s_1, \text{block } z \text{ in } c \Downarrow s_2}$	$\frac{S; O - z \vdash s_1, c \Downarrow s_2}{S; O \vdash s_1, \text{block}/1 z \text{ in } c \Downarrow s_2}$
$\frac{}{S; O \vdash s, \text{throw } e \Uparrow e, s}$	
$\frac{S; O_1 \vdash s_1, c_1 \Uparrow e, s_2 \quad S; O_2 \vdash s_2, c_2 \Downarrow s_3}{S; O_1 * O_2 \vdash s_1, \text{try } c_1 \text{ handle } e \text{ by } c_2 \Downarrow s_3}$	$\frac{S; O \vdash s_1, c_1 \Downarrow s_2}{S; O \vdash s_1, \text{try } c_1 \text{ handle } e \text{ by } c_2 \Downarrow s_2}$
$\frac{S; O_1 \vdash s_1, c_1 \Uparrow e, s_2 \quad S; O_2 \vdash s_2, c_2 \Uparrow e_2, s_3}{S; O_1 * O_2 \vdash s_1, \text{try } c_1 \text{ handle } e \text{ by } c_2 \Uparrow e_2, s_3}$	$\frac{S; O \vdash s_1, c_1 \Uparrow e_2, s_2 \quad e_2 \neq e}{S; O \vdash s_1, \text{try } c_1 \text{ handle } e \text{ by } c_2 \Uparrow e_2, s_2}$
$\frac{s \vdash E \Downarrow v}{S; O \vdash s, x := E \Downarrow s[x \mapsto v]}$	$\frac{S; O_1 \vdash s_1, c_1 \Downarrow s_2 \quad S; O_2 \vdash s_2, c_2 \Downarrow s_3}{S; O_1 * O_2 \vdash s_1, (c_1; c_2) \Downarrow s_3}$
$\frac{S; O \vdash s_1, c_1 \Downarrow s_2 \quad S(z) = c_2 \quad \emptyset; \emptyset \vdash s_2, c_2 \Downarrow s_3}{S; O \vdash s_1, c_1 \Downarrow s_3}$	$\frac{S; O - z \vdash s_1, c_1 \Downarrow s_2 \quad O(z) = c_2 \quad \emptyset; \emptyset \vdash s_2, c_2 \Downarrow s_3}{S; O \vdash s_1, c_1 \Downarrow s_3}$
$\frac{S(z) = c_2 \quad \emptyset; \emptyset \vdash s_1, c_2 \Downarrow s_2 \quad S; O \vdash s_2, c_1 \Downarrow s_3}{S; O \vdash s_1, c_1 \Downarrow s_3}$	$\frac{O(z) = c_2 \quad \emptyset; \emptyset \vdash s_1, c_2 \Downarrow s_2 \quad S; O - z \vdash s_2, c_1 \Downarrow s_3}{S; O \vdash s_1, c_1 \Downarrow s_3}$

Figure 3: Big-step semantics rules for exceptions and signal handling

$$\begin{array}{c}
O_1[z \mapsto c_h](z) = c_h \quad \emptyset; \emptyset \vdash s_1, c_h \Downarrow s_2 \quad S; O_1 - z \vdash s_2, c_1 \Downarrow s_3 \\
\hline
S; O_1[z \mapsto c_h] \vdash s_1, c_1 \Downarrow s_3 \quad S; O_2 \vdash s_3, c_2 \Downarrow s_4 \\
\hline
S; (O_1 * O_2)[z \mapsto c_h] \vdash s_1, (c_1; c_2) \Downarrow s_4 \\
\hline
S; O_1 * O_2 \vdash s_1, \text{bind}/1z \text{ to } c_h \text{ in } (c_1; c_2) \Downarrow s_4
\end{array}$$

Figure 4: Splitting of the  $O$  binding in seq. composed commands

Some auxiliary definitions will be required for the operational semantics. For a partial function  $f$ , we write  $f[x \mapsto v]$  for the function that maps  $x$  to  $v$  and coincides with  $f$  on all other arguments. In particular, we use this notation for updating states or signal bindings. We write  $\text{dom}(f)$  for the domain of definition of a partial function. For  $x \in \text{dom}(f)$ , we write  $f - x$  for the restriction of  $f$  to  $(\text{dom}(f) \setminus \{x\})$ . A signal binding is a finite partial function from signal names  $z$  to commands  $c$ . We will need a partial operation on signal bindings. In fact, this definition is the same as the separating conjunction from separation logic [13].

**Definition 3.2** Given two signal bindings  $O_1$  and  $O_2$ , we define a partial operation  $*$  as follows:

- If  $\text{dom}(O_1) \cap \text{dom}(O_2) = \emptyset$ , we write  $O_1 * O_2$  for  $O_1 \cup O_2$ .
- If  $\text{dom}(O_1) \cap \text{dom}(O_2) \neq \emptyset$ , then  $O_1 * O_2$  is undefined.

It is this splitting of a signal binding, analogous to the heap-splitting of separation logic, that gives one-shot behaviour to signals. Specifically, in a sequential composition  $(c_1; c_2)$ , the one-shot signals are split non-deterministically between the commands  $c_1$  and  $c_2$ . Moreover, every time a one-shot signal arrives and is handled, it is removed from the one-shot binding  $O$ . Thus, a one-shot signal may never be handled twice.

**Definition 3.3** Given two signal bindings  $S$  and  $O$ , the form of a big-step judgement is either

$$S; O \vdash s_1, c \Downarrow s_2$$

for normal termination, or

$$S; O \vdash s_1, c \Uparrow e, s_2$$

for exception throwing. The rules are given in Figure 3. The exception convention is assumed implicitly.

## 4 Examples

We examine how signal and exception handling interact in a series of examples, and discuss the question of priority between them.

### Examples for signals

The aim of the Figure 4 and Figure 5 is to show how one-shot and persistent signal bindings are "shared" between sequentially composed commands, and highlight the core difference between them (splitting versus copying).

In Figure 4, the one-shot signal binding  $O = O_1 * O_2$  (Definition 3.2) is split non-deterministically between commands  $c_1$  and  $c_2$ . When the new signal  $z$  is registered, it becomes an element of the domain  $(O_1 * O_2)[z \mapsto c_h]$ . However, is  $z \in \text{dom}(O_1[z \mapsto c_h])$  or  $z \in \text{dom}(O_2[z \mapsto c_h])$  will be determined

$$\begin{array}{c}
\frac{S[z \mapsto c_h](z) = c_h \quad \emptyset; \emptyset \vdash s_1, c_h \Downarrow s_2 \quad S[z \mapsto c_h]; O \vdash s_2, c_1 \Downarrow s_3}{S[z \mapsto c_h]; O \vdash s_1, c_1 \Downarrow s_3} \mathcal{D} \\
\hline
\frac{S[z \mapsto c_h]; O \vdash s_1, (c_1; c_2) \Downarrow s_6}{S; O \vdash s_1, \text{bind } z \text{ to } c_h \text{ in } (c_1; c_2) \Downarrow s_6} \\
\\
\mathcal{D} = \frac{S[z \mapsto c_h](z) = c_h \quad \emptyset; \emptyset \vdash s_3, c_h \Downarrow s_4 \quad \mathcal{F}}{S[z \mapsto c_h]; O \vdash s_3, c_2 \Downarrow s_6} \\
\\
\mathcal{F} = \frac{S[z \mapsto c_h](z) = c_h \quad \emptyset; \emptyset \vdash s_4, c_h \Downarrow s_5 \quad S[z \mapsto c_h]; O \vdash s_5, c_2 \Downarrow s_6}{S[z \mapsto c_h]; O \vdash s_4, c_2 \Downarrow s_6}
\end{array}$$

Figure 5: Multiple persistent signal handling in seq. composed commands

$$\begin{array}{c}
\frac{O[z \mapsto c_h](z) = c_h \quad \emptyset; \emptyset \vdash s_1, c_h \Downarrow s_2 \quad S; O - z \vdash s_2, c \Downarrow s_3}{S; O[z \mapsto c_h] \vdash s_1, c \Downarrow s_3} \\
\hline
S; O \vdash s_1, \text{bind}/1 z \text{ to } c_h \text{ in } c \Downarrow s_3
\end{array}$$

Figure 6: One-shot signal handling before the command

during the run time only. In this particular example, the signal  $z$  arrives in "scope" of the command  $c_1$  ( $z \in \text{dom}(O_1[z \mapsto c_h])$ ) and the bound handler runs. According to the one-shot signal binding nature, the binding for  $z$  is removed from  $O_1[z \mapsto c_h]$  and consequently from  $(O_1 * O_2)[z \mapsto c_h]$  as  $O_1[z \mapsto c_h] \subseteq (O_1 * O_2)[z \mapsto c_h]$ . Therefore,  $z \notin \text{dom}(O_2)$  and if the signal  $z$  arrives during the execution of the command  $c_2$ , it will be ignored.

In Figure 5, we focus on a persistent signal binding. The key difference with the one-shot binding is that the binding is just copied to the every command without splitting or modification. Thus, the same signal handler may run any number of times during the execution of the commands  $c_1$  and  $c_2$ . This behaviour is possible because triggering a persistent signal handler does not remove the corresponding binding.

### Examples for signals and exceptions

Suppose that a signal handler relies on some resource (valid pointer, open socket, active connection, etc.) available in the particular scope. However, as a side effect of the handler execution, the resource becomes unavailable (freed pointer, closed socket, inactive connection). In this situation, multiple handler executions may lead to the program fail and abrupt termination.

Obviously, one-shot signal handlers are perfectly fit for purpose. In Figure 6, the one-shot signal handler  $c_h$  runs before the command  $c$ . Thus, when control flow returns to  $c$ , the one-shot signal binding no longer contains a binding for the handler  $c_h$ . In Figure 7, the one-shot signal handler  $c_h$  runs after the command  $c$ , and at that point the signal binding no longer contains a binding for  $c_h$ . Note that the signal handlers (persistent and one-shot) that are still bound might be triggered if the corresponding signal arrives after the  $c_h$ .

On the other hand, a persistent handler combined with an exception imitates one-shot signal handlers





$$\begin{array}{c}
\frac{S[z \mapsto h](z) = h \quad \emptyset; \emptyset \vdash s_1, h \Downarrow s_2 \quad \overline{S[z \mapsto h]; O \vdash s_2, \text{throw } e \Uparrow e, s_2}}{S[z \mapsto h]; O \vdash s_1, \text{throw } e \Uparrow e, s_2} \\
\hline
\frac{S; O \vdash s_1, (\text{bind } z \text{ to } h \text{ in } \text{throw } e) \Uparrow e, s_2 \quad S; O \vdash s_2, g \Downarrow s_3}{S; O \vdash s_1, \text{try } (\text{bind } z \text{ to } h \text{ in } \text{throw } e) \text{ handle } e \text{ by } g \Downarrow s_3}
\end{array}$$

Figure 10: Example of the derivation tree: a signal binding inside an exception block

$$\begin{array}{c}
\frac{S[z \mapsto h](z) = h \quad \emptyset; \emptyset \vdash s_1, h \Downarrow s_2 \quad \overline{S[z \mapsto h]; O \vdash s_2, \text{throw } e \Uparrow e, s_2}}{S[z \mapsto h]; O \vdash s_1, \text{throw } e \Uparrow e, s_2} \quad \mathcal{F} \\
\hline
\frac{S[z \mapsto h]; O \vdash s_1, \text{try } (\text{throw } e) \text{ handle } e \text{ by } g \Downarrow s_4}{S; O \vdash s_1, \text{bind } z \text{ to } h \text{ in } (\text{try } (\text{throw } e) \text{ handle } e \text{ by } g) \Downarrow s_4} \\
\hline
\mathcal{F} = \frac{S[z \mapsto h](z) = h \quad \emptyset; \emptyset \vdash s_2, h \Downarrow s_3 \quad S[z \mapsto h]; O \vdash s_3, g \Downarrow s_4}{S[z \mapsto h]; O \vdash s_2, g \Downarrow s_4}
\end{array}$$

Figure 11: Derivation tree for the combined signals and exceptions

### Interaction between signal and exception handling

There is potentially a pitfall in combining signals and jumps (such as exceptions), in that a jump could prevent a handler from being correctly uninstalled at the end of its scope. In fact the problem is quite general, and arises whenever resource management is combined with jumping. In our language as defined in Definition 3.1, such a potential problem case is presented by the following code:

`try (bind  $z$  to  $h$  in  $\text{throw } e$ ) handle  $e$  by  $g$`

The intended meaning is that the signal  $z$  is bound locally inside the body of an exception block. The signal handler may run immediately before the `throw  $e$`  command. However, once the exception has propagated to the exception handler, it has left the scope of the signal binding, so that the signal handler should not be able to run. To see that the big-step semantics (Figure 3) correctly handles this case, consider the derivation tree in Figure 10.

In a big-step semantics, block structure is handled correctly "for free". The extended signal binding  $S[z \mapsto h]$  is confined to the subtree of the body of the binding. When the body is left, the evaluation is resumed with the old  $S$ , which is what is used in the evaluation of  $g$ . Even when control leaves the signal block abruptly via an exception, there is no danger that the signal handler escapes from its scope. By contrast, in a small-step semantics (e.g.: abstract machine) the uninstalling of signal handlers needs to be performed explicitly.

### The question of priority

In our operational semantics, exception propagation has higher priority than exception handling. Thus, a signal might be handled only before the exception has been *thrown* and after it has been caught (Figure 11). The command `throw` does not change the state itself, thus the state remains unchanged until

$$\begin{array}{c}
\frac{S[z \mapsto h]; O \vdash s_1, \text{throw } e \uparrow e, s_1 \quad S[z \mapsto h](z) = h \quad \emptyset; \emptyset \vdash s_1, h \Downarrow s_2}{S[z \mapsto h]; O \vdash s_1, \text{throw } e \uparrow e, s_2} \\
\hline
\frac{S; O \vdash s_1, (\text{bind } z \text{ to } h \text{ in } \text{throw } e) \uparrow e, s_2 \quad S; O \vdash s_2, g \Downarrow s_3}{S; O \vdash s_1, \text{try } (\text{bind } z \text{ to } h \text{ in } \text{throw } e) \text{ handle } e \text{ by } g \Downarrow s_3}
\end{array}$$

Figure 12: Signal handler runs after the throw

the exception is caught, when there are different options: if no signal arrives then the exception handler runs, or else the signal handler runs first, and only then the exception handler proceeds.

However, one can design an implementation where signal handling has higher priority. Thus, a signal handler should be processed even if exception propagation takes place (Figure 12). In a semantics with signal priority, the state is changed by the signal handler even during the exception propagation. One can make a few interesting observation about it. During exception propagation, control flow exits nested blocks, which in turn may have different signal bindings. Thus, depending in which block a signal arrives, the corresponding handler will interrupt the exception propagation. In addition, it might be the case that the signal is blocked in that scope, thus propagation would not be interrupted.

## 5 Stack machine for signal handlers

We define an abstract machine in order to highlight some of the issues that may arise in possible implementations of block-structured signals, such as managing the stack. The implementation of signal handlers in our abstract machine was inspired by the real implementations of exceptions in contrast to the unstructured `longjmp` that exceptions were designed to replace.

The defined block-structured form of signal handling requires a signal handler to be installed at the beginning of the block and uninstalled at the end. Therefore, to keep track of signal handlers in a particular scope, we use a signal stack. However, the addition of exceptions complicates the scoping of signal handlers. When control leaves a signal scope via a raised exception, the handler should be uninstalled. Thus, to implement the desired interaction between signal and exception scope, we keep track of signal handlers and exception handlers on the same stack. When an exception is raised, the stack is popped until the nearest enclosing handler for the exception name is found. The same popping of the common handler stack also removes any intervening signal handlers.

A machine configuration is of the form  $\langle c \mid s \mid \beta \mid J \mid K \rangle$ , where  $c$  is the expression that the machine is currently trying to evaluate,  $s$  is a state. The bit vector component  $\beta$  is used for keeping track of installed (not blocked) signals.  $J$  is a stack, which holds the signal and exception bindings.  $K$  is a continuation, which tells the machine what to do when it is finished with the current command  $c$ . The initial continuation is a special instruction `return`. The special symbol  $\blacksquare$  is used to represent an empty stack in the components  $J$  and  $K$ . When we get  $\langle \text{return} \mid s \mid \beta \mid \blacksquare \mid \blacksquare \rangle$ , program execution is finished. The full list of transition steps is given in Figure 13. To evaluate expression  $E$  in a state  $s$ , we apply the function *eval* (Definition 5.1), which returns a value  $v$ .

$\beta^0$  stands for a null bit vector (which means blocking or ignoring of all signals). The system instruction `pop-upd( $\beta'$ )` removes the top element from a stack  $J$  and updates  $\beta$  to  $\beta'$ . The system instruction `update( $\beta'$ )` updates  $\beta$  to  $\beta'$ . We define  $J$  as a data structure that follows stack discipline except in the

$$\begin{aligned}
\langle c_1; c_2 \mid s_1 \mid \beta_1 \mid J_1 \mid K_1 \rangle &\rightsquigarrow \langle c_1 \mid s_1 \mid \beta_1 \mid J_1 \mid c_2; K_1 \rangle \\
\langle x := E \mid s_1 \mid \beta_1 \mid J_1 \mid c'; K_1 \rangle &\rightsquigarrow \langle c' \mid s_1[x \mapsto v] \mid \beta_1 \mid J_1 \mid K_1 \rangle \\
&\text{where } \text{eval}(E, s_1) = v \\
\langle \text{bind}z\text{to}h\text{inc} \mid s \mid \beta \mid J \mid K \rangle &\rightsquigarrow \langle c \mid s \mid \beta + z \mid (z, h), J \mid \text{pop-upd}(\beta); K \rangle \\
\langle \text{bind}/1z\text{to}h\text{inc} \mid s \mid \beta \mid J \mid K \rangle &\rightsquigarrow \langle c \mid s \mid \beta + z \mid (z, h, 0), J \mid \text{pop-upd}(\beta); K \rangle \\
\langle \text{pop-upd}(\beta_1) \mid s \mid \beta_2 \mid (z, h), J \mid c; K \rangle &\rightsquigarrow \langle c \mid s \mid \beta_1 \mid J \mid K \rangle \\
\langle c \mid s \mid \beta \mid J_1, (z, h), J_2 \mid K \rangle &\rightsquigarrow \langle h \mid s \mid \beta^0 \mid J_1, (z, h), J_2 \mid \text{update}(\beta); c; K \rangle \\
&\text{handling of the persistent signal} \\
\langle c \mid s \mid \beta \mid J_1, (z, h, 0), J_2 \mid K \rangle &\rightsquigarrow \langle h \mid s \mid \beta^0 \mid J_1, (z, h, 1), J_2 \mid \text{update}(\beta - z); c; K \rangle \\
&\text{handling of the one-shot signal} \\
\langle \text{block}z\text{inc} \mid s \mid \beta_1 \mid J \mid K \rangle &\rightsquigarrow \langle c \mid s \mid \beta_1 - z \mid J \mid \text{update}(\beta_1); K \rangle \\
\langle \text{block}/1z\text{inc} \mid s \mid \beta_1 \mid J \mid K \rangle &\rightsquigarrow \langle c \mid s \mid \beta_1 - z \mid J \mid \text{update}(\beta_1); K \rangle \\
\langle \text{update}(\beta_1) \mid s \mid \beta_2 \mid J \mid c; K \rangle &\rightsquigarrow \langle c \mid s \mid \beta_1 \mid J \mid K \rangle \\
\langle \text{try } c_b \text{ handle } e \text{ by } h \mid s \mid \beta \mid J \mid K \rangle &\rightsquigarrow \langle c_b \mid s \mid \beta \mid (e, h), J \mid \text{pop-upd}(\beta); K \rangle \\
\langle \text{throw}e_1 \mid s \mid \beta \mid J_1, (e_1, h), J_2 \mid K_1 \rangle &\rightsquigarrow \langle h \mid s \mid \beta' \mid J_2 \mid K_2 \rangle \\
&\text{where } \text{unwind}(e_1, (J_1, (e_1, h), J_2), K_1) = (h, \beta', J_2, K_2).
\end{aligned}$$

Figure 13: Transition steps

case of one-shot signal handling. The  $J$  stack is manipulated by the system instructions that are pushed in and popped out from the continuation stack  $K$ .

$\beta$  is a function from signal names  $z$  to Booleans. For each signal name  $z$ ,  $\beta(z)$  tells us whether the signal is currently enabled. Then  $\beta + z$  is a shorthand for  $\beta[z \mapsto \text{true}]$  and  $\beta - z$  stands for  $\beta[z \mapsto \text{false}]$ .

For a  $\text{throw}e_1$  command, where  $e_1 \in \text{dom}(J)$ , we apply the  $\text{unwind}$  function (Definition 5.2), which returns a quadruple that is used to construct the next machine configuration. If  $e_1 \notin \text{dom}(J)$ , then the machine gets stuck with an unhandled exception, in the sense that there is no transition for this configuration, so that

$$\langle \text{throw}e_1 \mid s \mid \beta \mid J \mid K \rangle \not\rightsquigarrow$$

An exception binding tag has the form of  $(e, h)$ , where  $e$  is an exception identifier, and  $h$  is a handler. A persistent signal binding tag has the form of  $(z, h)$ , where  $z$  is a signal name, and  $h$  is a handler. A one-shot signal binding tag has the form of  $(z, h, u)$ , where  $z$  is a signal name,  $h$  is a handler, and  $u$  is a bit indicating that the handler has been used once ( $u=1$ ) or not ( $u=0$ ). Handling of the one-shot signals requires update of the  $J$  stack; to be more precise, the bit  $u$  in  $(z, h, u)$  is updated.

**Definition 5.1 (eval function)**

$$\begin{aligned}\text{eval}(x, s) &= s(x) \\ \text{eval}(E_1 + E_2, s) &= \text{eval}(E_1, s) + \text{eval}(E_2, s)\end{aligned}$$

**Definition 5.2 (unwind function)**

$$\begin{aligned}\text{unwind}(e_1, J, c; K) &= \text{unwind}(e_1, J, K) \\ \text{unwind}(e_1, J, \text{update}(\beta); K) &= \text{unwind}(e_1, J, K) \\ \text{unwind}(e_1, ((z, h), J), \text{pop-upd}(\beta); K) &= \text{unwind}(e_1, J, K) \\ \text{unwind}(e_1, ((e_1, h), J), \text{pop-upd}(\beta); K) &= (h, \beta, J, K)\end{aligned}$$

**Implementation of signals**

We compare how our idealized stack machine models features of real signal implementations.

**Bit vector** In our machine,  $\beta$  stands for the bit vector of installed not currently blocked signals; and  $\beta^0$  stands for a null bit vector that may be interpreted as "all signals are blocked" or "no signals are installed". The use of this bit vector almost directly corresponds to the bit maps used in real implementations. In real implementations, every signal has a default pre-assigned handler. To imitate the same behaviour, in our implementation it is possible to run a command inside of nested blocks in which all signals are bound to their default handlers.

**Exceptions and signals** In real implementations (as explained in [4], ISO/IEC 14882 [8, 9]), exception throwing inside of signal handlers is not recommended, due to implementation restrictions. Moreover, the existing implementation of signals is not block structured. By contrast, our abstract machine and big-step semantics deal with block structured signals and allow signal handlers to throw exceptions.

**Implementation of exception handling** In real implementations (e.g.: Itanium [5], and as described in [10, 4, 3]), exception handling is implemented by use of stack unwinding. Exception handling in our implementation resembles handling in real implementations, except the fact that the abstract machine uses the extra stack  $J$  to keep track of block structures, and the  $J$  is manipulated by special instructions in the continuation  $K$ .

## 6 Examples of the machine runs

We have already seen in previous examples (e.g.: Figure 8 and Figure 11) that the big-step semantics gives us block structure for free. This becomes very useful in studying block structured constructs and their interactions. By contrast, the machine needs to manage block structure explicitly with a help of the stack. The examples of corresponding machine runs are given in Figure 14 and Figure 15. Please note, the  $\text{pop-upd}(\beta^0)^2$  stands for  $\text{pop-upd}(\beta^0); \text{pop-upd}(\beta^0)$ .

The example in Figure 4 shows how the big-step syntax makes it easy to address one-shot signals with splitting the bindings. On the contrary, the machine needs to perform extra administrative work with the binding tags and the stack to implement one-shot signal handling (Figure 16).

One may observe that the abstract machine is more complex than the big-step semantics, as machine needs to deal with many details explicitly. Overall, we see that the machine is closer to implementations, whereas the big-step semantics is more convenient for abstract reasoning.

$$\begin{aligned}
& \langle \text{try}(\text{bindzto}(h; \text{throwe}) \text{ in } c) \text{ handle } e \text{ by } g \mid s_1 \mid \beta^0 \mid \blacksquare \mid \text{return} \rangle \\
\rightsquigarrow & \langle \text{bindzto}(h; \text{throwe}) \text{ in } c \mid s_1 \mid \beta^0 \mid (e, g) \mid \text{pop-upd}(\beta^0) \rangle \\
\rightsquigarrow & \langle c \mid s_1 \mid \beta^0 + z \mid (z, (h; \text{throwe})), (e, g) \mid \text{pop-upd}(\beta^0)^2 \rangle \\
\rightsquigarrow & \langle (h; \text{throwe}) \mid s_1 \mid \beta^0 \mid (z, (h; \text{throwe})), (e, g) \mid \text{update}(\beta^0 + z); c; \text{pop-upd}(\beta^0)^2 \rangle \\
\rightsquigarrow & \langle h \mid s_1 \mid \beta^0 \mid (z, (h; \text{throwe})), (e, g) \mid \text{throwe}; \text{update}(\beta^0 + z); c; \text{pop-upd}(\beta^0)^2 \rangle \\
\rightsquigarrow & \langle \text{throwe} \mid s_2 \mid \beta^0 \mid (z, (h; \text{throwe})), (e, g) \mid \text{update}(\beta^0 + z); c; \text{pop-upd}(\beta^0)^2 \rangle \\
\rightsquigarrow & \langle g \mid s_2 \mid \beta^0 \mid \blacksquare \mid \text{return} \rangle \\
\rightsquigarrow & \langle \text{return} \mid s_3 \mid \beta^0 \mid \blacksquare \mid \blacksquare \rangle
\end{aligned}$$

Figure 14: Binding inside of the try block

$$\begin{aligned}
& \langle \text{bindzto } h \text{ in } (\text{try}(\text{throwe}) \text{ handle } e \text{ by } g) \mid s_1 \mid \beta^0 \mid \blacksquare \mid \text{return} \rangle \\
\rightsquigarrow & \langle \text{try}(\text{throwe}) \text{ handle } e \text{ by } g \mid s_1 \mid \beta^0 + z \mid (z, h) \mid \text{pop-upd}(\beta^0) \rangle \\
\rightsquigarrow & \langle \text{throwe} \mid s_1 \mid \beta^0 + z \mid (e, g), (z, h) \mid \text{pop-upd}(\beta^0 + z); \text{pop-upd}(\beta^0) \rangle \\
\rightsquigarrow & \langle h \mid s_1 \mid \beta^0 \mid (e, g), (z, h) \mid \text{update}(\beta^0 + z); \text{throwe}; \text{pop-upd}(\beta^0 + z); \text{pop-upd}(\beta^0) \rangle \\
\rightsquigarrow & \langle \text{update}(\beta^0 + z) \mid s_2 \mid \beta^0 \mid (e, g), (z, h) \mid \text{throwe}; \text{pop-upd}(\beta^0 + z); \text{pop-upd}(\beta^0) \rangle \\
\rightsquigarrow & \langle \text{throwe} \mid s_2 \mid \beta^0 + z \mid (e, g), (z, h) \mid \text{pop-upd}(\beta^0 + z); \text{pop-upd}(\beta^0) \rangle \\
\rightsquigarrow & \langle g \mid s_2 \mid \beta^0 + z \mid (z, h) \mid \text{pop-upd}(\beta^0) \rangle \\
\rightsquigarrow & \langle h \mid s_2 \mid \beta^0 \mid (z, h) \mid \text{update}(\beta^0 + z); g; \text{pop-upd}(\beta^0) \rangle \\
\rightsquigarrow & \langle \text{update}(\beta^0 + z) \mid s_3 \mid \beta^0 \mid (z, h) \mid g; \text{pop-upd}(\beta^0) \rangle \\
\rightsquigarrow & \langle g \mid s_3 \mid \beta^0 + z \mid (z, h) \mid \text{pop-upd}(\beta^0) \rangle \\
\rightsquigarrow & \langle \text{pop-upd}(\beta^0) \mid s_4 \mid \beta^0 + z \mid (z, h) \mid \text{return} \rangle \\
\rightsquigarrow & \langle \text{return} \mid s_4 \mid \beta^0 \mid \blacksquare \mid \blacksquare \rangle
\end{aligned}$$

Figure 15: Exception handling inside of the binding

$$\begin{aligned}
& \langle \text{bind/1zto } c_h \text{ in } (c_1; c_2) \mid s_1 \mid \beta^0 \mid \blacksquare \mid \text{return} \rangle \\
\rightsquigarrow & \langle c_1; c_2 \mid s_1 \mid \beta^0 + z \mid (z, h_1, 0) \mid \text{pop-upd}(\beta^0); \text{return} \rangle \\
\rightsquigarrow & \langle c_1 \mid s_1 \mid \beta^0 + z \mid (z, h_1, 0) \mid c_2; \text{pop-upd}(\beta^0); \text{return} \rangle \\
\rightsquigarrow & \langle h_1 \mid s_1 \mid \beta^0 \mid (z, h_1, 1) \mid \text{update}(\beta^0); c_1; c_2; \text{pop-upd}(\beta^0); \text{return} \rangle \\
\rightsquigarrow & \langle \text{update}(\beta^0) \mid s_2 \mid \beta^0 \mid (z, h_1, 1) \mid c_1; c_2; \text{pop-upd}(\beta^0); \text{return} \rangle \\
\rightsquigarrow & \langle c_1 \mid s_2 \mid \beta^0 \mid (z, h_1, 1) \mid c_2; \text{pop-upd}(\beta^0); \text{return} \rangle \\
\rightsquigarrow & \langle c_2 \mid s_3 \mid \beta^0 \mid (z, h_1, 1) \mid \text{pop-upd}(\beta^0); \text{return} \rangle \\
\rightsquigarrow & \langle \text{pop-upd}(\beta^0) \mid s_4 \mid \beta^0 \mid (z, h_1, 1) \mid \text{return} \rangle \\
\rightsquigarrow & \langle \text{return} \mid s_4 \mid \beta^0 \mid \blacksquare \mid \blacksquare \rangle
\end{aligned}$$

Figure 16: Signal binding and seq. composed commands

## 7 Conclusions

The present paper idealizes signal handling in combination with the more familiar exception handling to focus on some of their semantic and logical features. The semantics of one-shot handlers is reminiscent of linearly-used continuations [2] and the resource usage in separation logic [13]. The way we have treated signal bindings in the big-step semantics borrows ideas from linear logic. Recall that we write

$$S; O \vdash s_1, c \Downarrow s_2$$

for a judgement involving a persistent signal binding  $S$  and a one-shot signal binding  $O$ . As we have illustrated with the examples in Section 4, the signal binding  $S$  can be shared between two commands  $c_1$  and  $c_2$  in a sequential composition, whereas  $O$  has to be split into disjoint parts  $O_1$  and  $O_2$ . This splitting prevents a one-shot signal handler from being re-used and makes it a linear resource just like the contexts in a linear logic. In fact, Dual Intuitionistic Linear Logic [1] has two zones  $\Gamma$  and  $\Delta$  in the context, one which allows sharing and one which does not, as in the following rule that shares  $\Delta$  and splits  $\Gamma$ :

$$\frac{\Gamma_1; \Delta \vdash M : A \multimap B \quad \Gamma_2; \Delta \vdash N : A}{\Gamma_1, \Gamma_2; \Delta \vdash MN : B}$$

We are not aware of previous operational semantics for signals, although Feng, Shao, Guo and Dong [6] presents a program logic for assembly language with interrupts, which are analogous to signals at the hardware level.

Hutton and Wright [7] study interruptions as asynchronous exceptions. By contrast, signals are a software alternative to hardware interrupts, where signal handlers could be addressed as asynchronous subroutine calls.

Signals have been part of the long evolution of Unix, and are correspondingly complex. To implement block-structured signal handling and integrate it with exceptions, the present signal mechanism may have to be revisited. The present implementations pose severe restrictions on programmers, for instance on using non-local control in a handler. Removing such implementation restrictions would enable natural programming idioms. In further work, we hope to build on the operational semantics presented here for proving soundness of a Hoare logic for signals.

The formal connection between the big-step operational semantics and the signals abstract machine remains to be established. We conjecture that they are observationally equivalent and that this may be proved by way of a simulation relation.

## References

- [1] Andrew Barber & Gordon Plotkin (1998): *Dual Intuitionistic Linear Logic*. Technical Report, University of Edinburgh.
- [2] Josh Berdine, Peter W. O'Hearn, Uday Reddy & Hayo Thielecke (2002): *Linear Continuation Passing*. *Higher-order and Symbolic Computation* 15(2/3), pp. 181–208, doi:10.1023/A:1020891112409.
- [3] Daniel Bovet & Marco Cesati (2002): *Understanding the Linux Kernel, Second Edition*, 2 edition. O'Reilly & Associates, Inc., Sebastopol, CA, USA.
- [4] Christophe de Dinechin (2000): *C++ exception handling for IA-64*. In: *Proceedings of the 1st conference on Industrial Experiences with Systems Software - Volume 1*, WIESS'00, USENIX Association, Berkeley, CA, USA, pp. 8–8. Available at <http://dl.acm.org/citation.cfm?id=1251503.1251511>.

- [5] (2001): *Itanium C++ ABI: Exception Handling*. Available at <http://www.codesourcery.com/cxx-abi/abi-eh.html>. (Revision: 1.22).
- [6] Xinyu Feng, Zhong Shao, Yu Guo & Yuan Dong (2009): *Certifying Low-Level Programs with Hardware Interrupts and Preemptive Threads*. *J. Autom. Reasoning* 42(2-4), pp. 301–347, doi:10.1007/s10817-009-9118-9.
- [7] Graham Hutton & Joel Wright (2007): *What is the meaning of these constant interruptions?* *J. Funct. Program.* 17(6), pp. 777–792, doi:10.1017/S0956796807006363.
- [8] (2011): *ISO/IEC 14882:2011 Information technology - Programming languages - C++*. Available at <http://www.iso.org>.
- [9] (1999): *ISO/IEC 14882:1999 Programming languages - C++*. Available at [www.iso.ch](http://www.iso.ch).
- [10] Michael Kerrisk (2010): *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*, 1 edition. No Starch Press. Available at <http://www.worldcat.org/isbn/1593272200>.
- [11] Robin Milner, Mads Tofte, Robert Harper & David MacQueen (1997): *The Definition of Standard ML (Revised)*. MIT Press. Available at <http://www.worldcat.org/isbn/0262631814>.
- [12] Eugenio Moggi (1989): *Computational Lambda Calculus and Monads*. In: *Proceedings, Fourth Annual Symposium on Logic in Computer Science*, pp. 14–23, doi:10.1109/LICS.1989.39155.
- [13] John C. Reynolds (2002): *Separation Logic: A Logic for Shared Mutable Data Structures*. In: *Logic in Computer Science (LICS)*, IEEE, pp. 55–74, doi:10.1109/LICS.2002.1029817.
- [14] Kay Robbins & Steve Robbins (2003): *UNIX Systems Programming: Communication, Concurrency and Threads (2nd Edition)*. Prentice Hall PTR. Available at <http://www.worldcat.org/isbn/0130424110>.
- [15] Sandra Loosemore and Richard M. Stallman and Roland McGrath and Andrew Oram and Ulrich Drepper (2007): *The GNU C Library Reference Manual*, 0.12 edition. Available at <http://www.gnu.org/software/libc/manual/pdf/libc.pdf>. last updated 2007-10-27, for version 2.8.
- [16] Richard W. Stevens & Stephen A. Rago (2005): *Advanced Programming in the UNIX(R) Environment (2nd Edition)*. Addison-Wesley Professional. Available at <http://www.informit.com/store/product.aspx?isbn=0201433079>.